

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Compiling Source Code Using Generic Classes

Inventors:

Makarand Gadre

ATTORNEY'S DOCKET NO. MS1-1596US

EL996276242

Compiling Source Code Using Generic Classes

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to co-pending U.S. patent application Ser. No. _____, Attorney Docket No. MS1-1597US, entitled "Authoring and Using Generic Classes in JAVA Language Code" by Makarand Gadre and Pratap V. Lakshman; which is filed concurrently herewith, assigned to the assignee of the present application, and incorporated herein by reference for all that it teaches and discloses.

TECHNICAL FIELD

The subject matter disclosed herein relates generally to methods, devices and/or systems for compiling source code that uses generic classes.

BACKGROUND

Frameworks include class libraries that provide software developers with tools for developing, testing, using, and deploying software applications. Examples of two popular frameworks are the .NET™ Framework from Microsoft® Corporation of Redmond, Washington, and the JAVA™ language framework from Sun Microsystems, Inc. of Palo Alto, California. Generic classes (in C++ referred to as template classes; also referred to as generic types) may be provided by such frameworks.

Generic classes refer to classes, interfaces and methods that operate uniformly on values of different types. Generic classes can speed software development by packaging classes, methods, and data and making them applicable

1 to multiple data types that are used frequently by developers. Generic classes are
2 useful because many common classes can be parameterized by the types of data
3 being stored and manipulated – these are called generic class declarations.
4 Similarly, many interfaces define contracts that can be parameterized by the types
5 of data they handle – these are called generic interface declarations. Methods may
6 also be parameterized by type in order to implement “generic algorithms”, and
7 these are known as ‘generic methods’.

8 A formal specification for a software language specifies standard syntax for
9 the language. Formal specifications for C++ and other languages set forth generic
10 class syntaxes that specify how generic classes (or, template classes) are defined
11 and declared; however, formal specifications for some languages, such as JAVA™
12 language, do not specify generic classes. Thus, generic classes that may be
13 provided in frameworks, or other software packages, are not readily accessible by
14 developers of JAVA™ language source code. For example, currently, JAVA™
15 language source code cannot use a generic class that may be provided by the
16 .NET™ Framework. Thus, to take full advantage of a framework, developers
17 need the capabilities for authoring, using, and compiling generic classes that may
18 be provided by the framework.

19 SUMMARY

21 Implementations described herein provide methods and systems for
22 compiling a generic class reference into an intermediate language executable by a
23 runtime engine. The generic class may be referenced in source code written in a
24 language for which use of generic classes is not formally specified.

1 An exemplary method includes receiving a portion of JAVA™ language
2 software having a declaration of an instance of a generic class, parsing the
3 declaration into a token corresponding to the generic class, and generating an
4 intermediate language code block corresponding to the parsed declaration. The
5 intermediate language code block is executable by a runtime engine.

6 An exemplary system for compiling includes a parser receiving JAVA™
7 language source code having an instruction referencing a generic class and
8 specifying a type of the generic class, and a code generator generating
9 intermediate language code representing the source code.

10 Additional features and advantages will be made apparent from the
11 following detailed description of illustrative embodiments, which proceeds with
12 reference to the accompanying figures.

13 14 **BRIEF DESCRIPTION OF THE DRAWINGS**

15 A more complete understanding of the various methods and arrangements
16 described herein, and equivalents thereof, may be had by reference to the
17 following detailed description when taken in conjunction with the accompanying
18 drawings wherein:

19 Fig. 1 is a block diagram generally illustrating an exemplary computer
20 system on which various exemplary technologies disclosed herein may be
21 implemented.

22 Fig. 2 is a block diagram illustrating an exemplary framework, a compiled
23 project and a runtime engine.
24
25

Fig. 3 is a block diagram illustrating an exemplary compiler operable to compile source code that references generic classes into project code executable by a runtime engine.

DETAILED DESCRIPTION

Turning to the drawings, wherein like reference numerals refer to like elements, various methods and converters are illustrated as being implemented in a suitable computing environment. Although not required, the methods and converters will be described in the general context of computer-executable instructions, such as program modules, being executed by a personal computer. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the methods and converters may be practiced with other computer system configurations, including hand-held devices, multi-processor systems, microprocessor based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The methods and converters may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Overview

Implementations described herein provide methods and systems for using generic classes in source code written in a language for which generic classes are

1 not formally specified. Generally, source code may be developed using a
2 framework wherein generic classes are available. For example, generic classes
3 associated with a framework capable of using multiple source codes and an
4 intermediate language, can be referenced in a JAVATM language. The source code
5 is converted into an intermediate language source code. Metadata can be
6 generated that describes any referenced generic classes.

7
8 Thus, an implementation enables a Visual J# .NETTM (VJ#TM) Compiler to
9 work with generic classes. In this regard, an improved VJ#TM compiler include
10 support for generic types, including data structures, information, and algorithms
11 that are processed and executed in connection with authoring and using generic
12 types. In one implementation, the VJ#TM compiler applies an algorithm of parsing
13 a variable or type declaration having references to generic classes, looking up
14 reference assemblies and validating types with respect to the generic classes,
15 utilizing data structures representing parsed and validated generics information,
16 and traversing a generic tree representation to generate common intermediate
17 language code.

20 **Exemplary Computing Environment**

21 Fig.1 illustrates an example of a suitable computing environment 120 with
22 which the subsequently described exemplary methods, compilers, parsers, etc.,
23 may be implemented.
24
25

1 Exemplary computing environment 120 is only one example of a suitable
2 computing environment and is not intended to suggest any limitation as to the
3 scope of use or functionality of the improved methods and arrangements described
4 herein. Neither should computing environment 120 be interpreted as having any
5 dependency or requirement relating to any one or combination of components
6 illustrated in computing environment 120.

7
8 The improved methods and arrangements herein are operational with
9 numerous other general purpose or special purpose computing system
10 environments or configurations. Examples of well known computing systems,
11 environments, and/or configurations that may be suitable include, but are not
12 limited to, personal computers, server computers, thin clients, thick clients, hand-
13 held or laptop devices, multiprocessor systems, microprocessor-based systems, set
14 top boxes, programmable consumer electronics, network PCs, minicomputers,
15 mainframe computers, distributed computing environments that include any of the
16 above systems or devices, and the like.

17
18 As shown in Fig. 1, computing environment 120 includes a general-purpose
19 computing device in the form of a computer 130. The components of computer
20 130 may include one or more processors or processing units 132, a system
21 memory 134, and a bus 136 that couples various system components including
22 system memory 134 to processor 132.

23
24 Bus 136 represents one or more of any of several types of bus structures,
25 including a memory bus or memory controller, a peripheral bus, an accelerated

1 graphics port, and a processor or local bus using any of a variety of bus
2 architectures. By way of example, and not limitation, such architectures include
3 Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA)
4 bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA)
5 local bus, and Peripheral Component Interconnects (PCI) bus also known as
6 Mezzanine bus.

7
8 Computer 130 typically includes a variety of computer readable media.
9 Such media may be any available media that is accessible by computer 130, and it
10 includes both volatile and non-volatile media, removable and non-removable
11 media.

12
13 In Fig. 1, system memory 134 includes computer readable media in the
14 form of volatile memory, such as random access memory (RAM) 140, and/or non-
15 volatile memory, such as read only memory (ROM) 138. A basic input/output
16 system (BIOS) 142, containing the basic routines that help to transfer information
17 between elements within computer 130, such as during start-up, is stored in ROM
18 138. RAM 140 typically contains data and/or program modules that are
19 immediately accessible to and/or presently being operated on by processor 132.

20
21 Computer 130 may further include other removable/non-removable,
22 volatile/non-volatile computer storage media. For example, Fig. 1 illustrates a
23 hard disk drive 144 for reading from and writing to a non-removable, non-volatile
24 magnetic media (not shown and typically called a "hard drive"), a magnetic disk
25 drive 146 for reading from and writing to a removable, non-volatile magnetic disk

1 148 (e.g., a "floppy disk"), and an optical disk drive 150 for reading from or
2 writing to a removable, non-volatile optical disk 152 such as a CD-ROM, CD-R,
3 CD-RW, DVD-ROM, DVD-RAM or other optical media. Hard disk drive 144,
4 magnetic disk drive 146 and optical disk drive 150 are each connected to bus 136
5 by one or more interfaces 154.

6
7 The drives and associated computer-readable media provide nonvolatile
8 storage of computer readable instructions, data structures, program modules, and
9 other data for computer 130. Although the exemplary environment described
10 herein employs a hard disk, a removable magnetic disk 148 and a removable
11 optical disk 152, it should be appreciated by those skilled in the art that other types
12 of computer readable media which can store data that is accessible by a computer,
13 such as magnetic cassettes, flash memory cards, digital video disks, random access
14 memories (RAMs), read only memories (ROM), and the like, may also be used in
15 the exemplary operating environment.

16
17 A number of program modules may be stored on the hard disk, magnetic
18 disk 148, optical disk 152, ROM 138, or RAM 140, including, e.g., an operating
19 system 158, one or more application programs 160, other program modules 162,
20 and program data 164.

21
22 The improved methods and arrangements described herein may be
23 implemented within operating system 158, one or more application programs 160,
24 other program modules 162, and/or program data 164.

1 A user may provide commands and information into computer 130 through
2 input devices such as keyboard 166 and pointing device 168 (such as a "mouse").
3 Other input devices (not shown) may include a microphone, joystick, game pad,
4 satellite dish, serial port, scanner, camera, etc. These and other input devices are
5 connected to the processing unit 132 through a user input interface 170 that is
6 coupled to bus 136, but may be connected by other interface and bus structures,
7 such as a parallel port, game port, or a universal serial bus (USB).

8
9 A monitor 172 or other type of display device is also connected to bus 136
10 via an interface, such as a video adapter 174. In addition to monitor 172, personal
11 computers typically include other peripheral output devices (not shown), such as
12 speakers and printers, which may be connected through output peripheral interface
13 175.

14
15 Logical connections shown in Fig. 1 are a local area network (LAN) 177
16 and a general wide area network (WAN) 179. The LAN 177 and/or the WAN 179
17 can be wired networks, wireless networks, or any combination of wired or wireless
18 networks. Such networking environments are commonplace in offices, enterprise-
19 wide computer networks, intranets, and the Internet.

20
21 When used in a LAN networking environment, computer 130 is connected
22 to LAN 177 via network interface or adapter 186. When used in a WAN
23 networking environment, the computer typically includes a modem 178 or other
24 means for establishing communications over WAN 179. Modem 178, which may
25

1 be internal or external, may be connected to system bus 136 via the user input
2 interface 170 or other appropriate mechanism.

3
4 Depicted in Fig. 1, is a specific implementation of a WAN via the Internet.
5 Here, computer 130 employs modem 178 to establish communications with at
6 least one remote computer 182 via the Internet 180.

7
8 In a networked environment, program modules depicted relative to
9 computer 130, or portions thereof, may be stored in a remote memory storage
10 device. Thus, e.g., as depicted in Fig. 1, remote application programs 189 may
11 reside on a memory device of remote computer 182. It will be appreciated that the
12 network connections shown and described are exemplary and other means of
13 establishing a communications link between the computers may be used.

14 15 **Exemplary Framework for Authoring, Using, and Compiling Generic Classes**

16 Fig. 2 shows an exemplary framework 200 and a compiled project 202
17 targeted for execution on a runtime engine (RE) 204. In object-oriented
18 programming, the terms “Virtual Machine” (VM) and “Runtime Engine” (RE)
19 have recently become associated with software that executes code on a processor
20 or a hardware platform. The RE 204 is operable to translate common intermediate
21 language code into microprocessor-specific binary that is executable by a
22 computer. In the description presented herein, the term “RE” includes VM. A RE
23 is often associated with a larger system (e.g., integrated development environment,
24 framework, etc.) that allows a programmer to develop an application.

1 For a programmer, the application development process usually involves
2 selecting a framework, coding in an object-oriented programming language
3 (OOPL) associated with that framework to produce a source code, and compiling
4 the source code using a compiler associated with the framework. In Fig. 2, the
5 framework 200 includes a code editor 206 for authoring (i.e., writing and/or
6 editing) project source code 208, project resources 210 (e.g., libraries, utilities,
7 etc.) and a compiler 212 for compiling the project source code 208. The
8 programmer may elect to save project source code and/or project resources in a
9 project file and/or a solution file, which may contain more than one project file. If
10 a programmer elects to compile project code and/or project resources, then the
11 resulting compiled code, and other information if required, is then typically made
12 available to users, e.g., as a compiled project, a solution, an executable file, an
13 assembly, etc.

14
15 The project resources 210 include class libraries 214 and other resources
16 216 (e.g., utilities, etc.). The class libraries 214 have definitions for classes that
17 may be used and/or authored by a developer. The classes contained in class
18 libraries 214 may have associated tokens for ease of referencing and compiling the
19 classes. For example, each class in the class libraries 214 can have a numerical
20 token that identifies the class.

21
22 One or more of the class definitions in the class libraries 214 correspond to
23 generic classes (also called generic types) (e.g., generic classes 314, Fig. 3). The
24 term “generic class” refers to classes, interfaces and methods that operate
25 uniformly on instances of different types and/or classes. By way of example, and

1 not limitation, a “Queue<Type>” class can be a generic class, wherein “Type” may
2 be declared as any of multiple allowable types or classes. The class library
3 definition of a generic class defines which types are allowable for the generic class
4 as well as the methods applicable to an instance of a generic class.

5
6 One or more standard generic classes may be provided by the framework
7 200. For example, a recently developed framework called the .NET™ framework
8 (Microsoft Corporation, Redmond, Washington) comes with a generic “Queue
9 <Type>” class, a “Stack <Type1>” class, a “Dictionary <Type1, Type2>” class,
10 and others. In addition, implementations of authoring methods and systems
11 described herein enable a developer define generic classes and make them
12 available in the class libraries 214 for use by the project code 208.

13
14 Precompiled data 218 shown in Fig. 2 includes any data created and/or used
15 by the compiler 212 to generate the compiled project 202. As is discussed in
16 further detail below, precompiled data 212 may include a parse tree 312 (Fig. 3), a
17 tokenized parse tree 316 (Fig. 3), and a validated tokenized parse tree 318 (Fig. 3).
18 Precompiled data includes various data structures and other information that are
19 intermediate between the source code 208 and the compiled project 202. Fig. 3
20 describes exemplary data and information in the precompiled data 218 and how
21 the compiler 212 uses the precompiled data to create the compiled project 202.

22
23 Fig. 2 shows a compiled project 202 generated by the compiler 212, which
24 includes portable code 220, metadata 222, and other data 224 (e.g., headers, native
25 image data, custom image data, etc.) that may be necessary for proper execution of

1 the portable code 220. The other data 224 may pertain to project resources 210 or
2 other resources. The compiled project 202 is typically available as one or more
3 files capable of distribution over a network. For example, the .NET™ framework
4 can produce a compiled project as a portable executable file containing
5 intermediate language code (IL code) and metadata, which is suitable for
6 distribution over the Internet and execution using the .NET™ Runtime Engine
7 (RE). In the .NET™ environment, the compiled project 214 may be referred to as
8 an assembly. Of course, one or more separate code files and one or more separate
9 data files may be contained within a project file or a compiled project file. Upon
10 receipt of the requisite file or files, a user can execute an embedded application or
11 applications on a RE associated with the selected framework. Fig. 2 shows the RE
12 204 associated with the framework 200.

13
14 Traditional frameworks, such as the JAVA™ language framework (Sun
15 Microsystems, Inc., Palo Alto, California), were developed initially for use with a
16 single object-oriented programming language (OOPL) (i.e., monolithic at the
17 programming language level); however, the .NET™ framework allows
18 programmers to code in a variety of OOPLs (e.g., VISUAL BASIC®, C++, Visual
19 C# .NET™, JScript, Visual J# .NET™, etc.). This multi-OOPL or multi-source
20 code framework is centered on a single compiled intermediate language having a
21 virtual object system (VOS).

22
23 The intermediate language (IL) generated by the .NET™ Framework is
24 often referred to as a “language-neutral” intermediate language because the IL
25 may be generated from software written in multiple source code languages. The

1 compiler 212 in a .NET™ Framework compiles all source code to a common IL,
2 irrespective of the source code language.

3
4 In contrast to the .NET™ Framework, other frameworks, such as the
5 JAVA™ language framework, do not allow programmers to code in a variety of
6 OOPLs. For example, the JAVA™ language framework requires that all source
7 code be in the JAVA™ language. The JAVA™ language framework compiles the
8 JAVA™ language source code into bytecodes, which are non-language-neutral.
9 Thus, in the JAVA™ language framework there cannot be bytecodes generated
10 from multiple OOPLs.

11
12 While the aforementioned .NET™ framework exhibits programming
13 language or source code interoperability, a need exists for methods, devices and/or
14 systems that allow authorship, use, and compilation of generic classes in a JAVA™
15 language project, solution, or source code. For example, a developer may want to
16 declare a predefined generic class in source code written in the JAVA™ language,
17 whereby the declared generic class is compiled into portable code. As further
18 described herein, exemplary methods, devices, and/or systems can facilitate
19 authoring, using, and compiling JAVA™ language source code in the .NET™
20 Framework.

21 22 **Implementing Generic .NET™ Classes in a JAVA™ Language**

23 With particular regard to the code editor 206, a user may author the project
24 source code 204 in a number of source code languages, including JAVA™, VJ++,
25 Visual J# .NET™, or other JAVA™ languages. As used herein, the term “JAVA™

1 language” refers to any source code language that is based on a formal JAVA™
2 language specification, such as, but not limited to, the JAVA™ Development Kit
3 (JDK™) 1.1.4. Although formal JAVA™ specifications do not specify generic
4 classes, exemplary implementations described herein provide ways for generic
5 classes to be authored, used and compiled in a JAVA™ language source code.

6
7 Implementations of methods and systems described herein enable authoring
8 generic classes in JAVA™ language source code for use by JAVA™ language
9 and/or software programs in other languages. In particular, these implementations
10 provide for authoring and using generic classes whereby instances of such generic
11 classes can be compiled into a common intermediate language (CIL) and executed
12 by a runtime engine, such as the runtime engine 204. A generic class may be
13 authored by defining the generic class such that methods and data of the generic
14 class are uniformly applicable to multiple different classes. In addition, such
15 generic classes authored in JAVA™ language may be used (e.g., declared,
16 referenced, etc.) by software programs written in other languages, such as C++
17 and Visual C# .NET™.

18
19 In a particular implementation, angular brackets are used in JAVA™
20 language source code to identify classes associated with a generic class. Between
21 the angular brackets, at least one unconstrained type or class is specified. The
22 following examples illustrate how a developer may author a generic class in
23 JAVA™ language source code.

24
25 Example 1:


```

1      public class MyGenericClass<X>
2      {
3          public MyGenericClass()
4          {
5              // constructor
6          }
7
8          public void Set(X xvar)
9          {
10             // code that may change state of this class
11         }
12
13         public X ReturnResult()
14         {
15             X xvar;
16             // code that may change xvar
17             return xvar;
18         }
19     }
20

```

21 Example 1 illustrates a generic class definition in JAVA™ language source
22 code in which the type argument, identified by 'X', can be of any class. The 'X'
23 class is called an unconstrained type because it can be of any class. The generic
24 class can be instantiated by providing a value for the type argument. In so doing, a
25 'constructed type' is created.

26 A second example of a generic class definition in JAVA™ language source
27 code is shown in example 2 shown below

28 Example 2:

```

29
30      public class MyGenericClass<X implements Comparable>
31      {
32          public MyGenericClass()
33          {
34              // constructor
35          }
36
37          public void Set(X xvar)
38          {
39              // code that may change state of this class
40          }
41      }
42

```

```

    }
1   public X ReturnResult()
2   {
3       X xvar;
4       // code that may change xvar
5       return xvar;
6   }
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

```

Example 2 illustrates how for certain generic classes each type-parameter may be qualified by an explicit-type-parameter-constraint. The specification of an explicit constraint is optional. If given, the constraint is a reference-type that specifies a minimal “type-bound” that every instantiation of the type parameter must support (for example, the constraint may be that the type parameter must implement a certain interface, inherit from a certain class, or provide a default constructor). In Example 2 above, the generic class can be instantiated by providing a value for the type argument, identified by ‘X’; the value provided must be of a class that implements the IComparable interface.

The foregoing examples illustrate how a developer may author generic classes in the JAVA™ language using the code editor 206. Such authored generic classes can be included in the generic classes of the class libraries 214. Other generic classes and types may be provided in the class libraries 214. As discussed earlier, such generic classes, whether or not they are authored in JAVA™ language, may be used by JAVA™ language programs and/or other non-JAVA™ language programs.

In a .NET™ Framework implementation, the generic classes (i.e., types), parameters, non-generic classes, and instantiated generic classes are defined by various code sections, such as .NET Assemblies, .NET Class Libraries, and User Code. A .NET™ Assembly is a collection of classes in MSIL form (e.g., classes available in .NET™ Frameworks). Table 1 illustrates an exemplary arrangement.

Table 1

Description	Defined By
.NET Generic Type	.NET Class Libraries, .NET Assembly
Formal Parameter Type to a .NET Generic Type	.NET Class Libraries, .NET Assembly
Type Parameter of Generic Type to be instantiated	User Code
Non Generic Type	.NET Class Libraries, .NET Assembly, User Code
Constructed Type	User Code

Thus, a .NET Class Library and/or a .NET Assembly contain definitions of generic classes, and specify the formal parameter types/classes that can be passed to a generic class. User code, such as project code 208 and user-authored class libraries, specifies any constructed types (i.e., instantiated generic classes).

Generic classes that have been defined and stored in the class libraries 214 can be used by developers, even in source code written in languages for which the use of generic classes has not been formally specified, through implementations described herein. For example, a developer can declare, or otherwise reference, a generic class in JAVA™ language source code. In the .NET™ framework, a developer can create JAVA™ language source code using Visual J# .NET™ that

1 includes declarations of instances of pre-defined generic classes. As is discussed
2 in further detail below, the compiler 212 is operable to compile declared instances
3 of generic classes into portable code 220 in languages that do not formally specify
4 use of generic classes.

5
6 With regard to using generic classes, a developer specifies in the source
7 code any unconstrained types or classes defined in the generic class definition. As
8 discussed above, when source code declares an instance of a generic class with an
9 allowable unconstrained type, the instance of the generic class is referred to as a
10 constructed class. A constructed class is a species of the generic class. For
11 example, if a generic class Queue, is defined as 'Queue<X>,' wherein class 'X' is
12 unconstrained, a declaration of 'Queue<int>' is referred to as a constructed class.

13
14 A developer specifies a constructed class of the desired generic class, and
15 then uses the constructed class much like other classes. The developer can declare
16 an instance of the constructed class, reference the instance of the constructed class,
17 apply operations or methods to the instance of the constructed class, and the like.

18
19 Examples of declared generic types are shown below in Table 2, in which
20 parameter 'X' refers to an unconstrained type:

21
22 **Table 2**

23 Declared Generic Type	Instantiated Type
24 Queue<X>	Queue <int> abc = new Queue <int>;
	Queue <System.String> abc = new
	Queue<System.String>;
25	Queue <Queue <System.String> > = new Queue <Queue

	<System.String> >;
1	Lookup<int, X> Lookup<int, Object> lu = new Lookup<int, Object>;
2	Lookup <int, Queue<String> > = new Lookup<int, Queue<String> >; // Nested Generic Type
3	class STR extends String
4	... SLookup<String, Object> slu = new SLookup<String, Object>;
5	SLookup<STR, Object> slu2 = new SLookup<STR, Object>;
6	// The next line would be error because // System.IntPtr is not an instanceof(String); SLookup<System.IntPtr, Object> = new SLookup<System.IntPtr, Object>;

Some generic classes may allow for nesting of classes. Nested classes refer to classes within classes. For example, a constructed class of the generic class 'Queue<X>' may be 'Queue<Queue<int>>,' wherein 'int' is a nested class; i.e., 'int' is nested in the inner 'Queue<>' generic class. In the foregoing example, because 'X' is unconstrained, generic classes can be nested at any number of levels. Thus, a constructed class takes the general form 'GC<GC<GC<...>>,' where 'GC' refers to the generic class. In a .NET™ implementation, nested classes may be used in JAVA™ language source code, and source code of other languages that may not formally specify use of generic classes.

Existing JAVA™ language source code can be easily adapted to use resources, such as generic classes, which may be provided by a framework or other software development package. In a framework environment, the adapted JAVA™ language source code can be compiled for execution by a runtime engine. A developer can modify existing source code to include references to generic classes. The developer simply needs to identify a generic class that is available from the framework or other software development package, and specify the class

1 (or classes) that are unconstrained parameters for the generic class using the
2 proper syntax. The developer creates a constructed class by declaring a generic
3 class specifying the unconstrained class (or classes) to be used. An instance of the
4 constructed class can then be declared and used.

5
6 For example, a JAVATM language source code developer may want to port
7 existing JAVATM language code to the .NETTM framework and use the generic
8 classes provided by .NETTM. The existing JAVATM language code may have been
9 written in standard JAVATM language or in a variation of JAVATM language such as
10 Visual J#TM, Jscript, or J++. Regardless of the original JAVATM language used,
11 Visual J# .NETTM in the .NETTM framework enables a developer to port the
12 existing JAVATM language code to the .NETTM framework and use the generic
13 classes of the .NETTM framework.

14 15 **Generating Executable Code From Source Code Using Generic Classes**

16 Compiling source code that uses generic classes involves generating a
17 compiled project representative of the source code. The compiled project is
18 readily executable by a microprocessor, using a runtime engine. The compiled
19 project may also be portable to various platforms, hardware, etc. A common
20 intermediate language (CIL) can facilitate portability of the compiled project.

21
22 Thus, one implementation of portable code 218 includes a common
23 intermediate language (CIL), such as Microsoft® Intermediate Language (MSIL)
24 code. MSIL defines a virtual instruction set. The MSIL is typically translated by
25 the runtime engine 222 into lower-level instructions executable by a

1 microprocessor. The MSIL is portable by virtue of the fact that the runtime engine
2 222 is microprocessor or platform aware. A particular implementation of the
3 framework 200 includes Visual J# .NET. Visual J# .NET includes an editor for
4 writing and editing source code using JAVATM language syntax, and a compiler for
5 compiling the JAVATM language source code into Microsoft ® Intermediate
6 Language (MSIL).

7
8 Fig. 3 is a block diagram illustrating an exemplary compiler 212
9 performing operation with respect to project code 208 to generate portable code
10 220 executable by a runtime engine. The compiler 212 includes a parser 302,
11 lexical analyzer (lexer) 304, common intermediate language (CIL) importer 306,
12 semantic analyzer 308, and code generator 310.

13
14 The parser 302 receives the project source code 208 or other input and
15 generates lexemes based on the source code 208. A lexeme is a minimal lexical
16 unit of a computing language, such as a keyword, identifier, literal, punctuation,
17 and the like, that is recognized by the lexer 304. Typically, the stream of
18 characters making up the source program 208 is read by the parser 302, one at a
19 time, and grouped into lexemes, which are passed to the lexer 304.

20
21 In one implementation, the parser 302 reads JAVATM language source code
22 from the project code 208, which includes references to generic classes 314. The
23 parser 302 divides a reference to a generic class into the generic class name, and
24 one or more associated classes, which may be constrained or unconstrained. For
25

1 example, if 'Queue<X>' is a generic class, a declaration 'Queue<int>' may be
2 divided into lexemes 'Queue' and 'int'.

3
4 The lexer 304 analyzes the syntax of the lexemes generated by the parser
5 302 with respect to a formal computing grammar. The lexer 304 resolves the
6 lexemes into identifiable parts before translation into lower level machine code.
7 The lexer 304 may also check to see that all input has been provided that is
8 necessary. During compilation, the lexer 304 issues an error if the lexemes cannot
9 be resolved to identifiable parts defined in the formal computing grammar.

10
11 In one implementation, the output of the lexer 304 is a parse tree 312. The
12 parse tree 312 is a representation of the source code 208 in which types referenced
13 in the project code 208 are separated in preparation for code generation. The parse
14 tree 312 may be a hierarchical, or tree, structure; in which parameters of generic
15 class declarations are listed under the generic class. Nested classes of a generic
16 class reference are presented at lower branches under the generic class. For
17 example, a line of Visual J# .NET™ (a JAVA™ language) source code
18 MyGenericClasses.LookupTable<long, MyGenericClasses.Queue<String> > may
19 be represented in the parse tree 312 as follows:

```
20 CType => MyGenericClasses.LookupTable  
21 ClassTree =>  
22     CType => long  
23     ClassTree => null  
24     CType => MyGenericClasses.Queue  
25     ClassTree =>  
26         CType => String  
27         ClassTree = null,
```


1 wherein 'LookupTable' is a generic class, having two parameters, in which
2 the second parameter is unconstrained as to type. In the above example, the
3 second parameter of the 'LookupTable' is 'Queue,' which is a generic class
4 having a nested class of 'String.' The parser interacts with the CIL importer 306
5 to validate direct references to the generic classes based on metadata that describes
6 the generic classes.

7
8 In an exemplary implementation, the lexer 304 constructs variables of type
9 CClass_Type from the project code 208. CClassType is a subclass of Class
10 CType. In this implementation, the parser 302 fills in recursive (i.e., nested)
11 CClass_Types for generic classes. Later, the lexer 304 traverses the tree while
12 validating each CType and obtaining an associated CClass object reference. When
13 the CClass object reference is created, the CIL importer 306 is called, which allots
14 a token to the CClass object. CClass_Type, CClass and CClass_Info objects are
15 kept unique for the duration of the compiler session.

16
17 Thus, the CIL importer 306 generates a tokenized parse tree 316 based on
18 the parse tree 312 and generic class definitions in the generic classes 314. The
19 generic classes 314 may be obtained from class libraries (e.g., class libraries 208,
20 Fig. 2) or other compiled projects (e.g., assemblies in .NET™). In the tokenized
21 parse tree 312, the types are represented as tokens that refer to defined types. For
22 example, a constructed class 'Queue<int, string>' may be represented in the parse
23 tree 312 as follows:

24 TokenCurrent
25 Token1
 Token2,

1 wherein "TokenCurrent" is a token associated with generic class 'Queue,' Token1
2 is a token associated with class 'int', and Token2 is a token associated with class
3 'string.'

4
5 A particular implementation of the CIL importer 306 also generates
6 metadata related to the classes referenced in the project code 208. The CIL
7 importer 306 gathers metadata from class definitions and populates the tokenized
8 parse tree 316 with the metadata.

9
10 In a .NET™ implementation of the CIL importer 306, the CIL importer
11 creates Microsoft® Intermediate Language (MSIL) assembly tokens using native
12 .NET™ Metadata Application Programming Interfaces (APIs). The CIL importer
13 306 uses the CClassType data created by the lexer 304 to construct data of type
14 CClass. CClass variables store ClassInfo, which include metadata descriptive of
15 the class. The CIL importer 306 stores the MSIL assembly tokens in data of type
16 CClassInfo. Every CClass_Type has a field to hold the CClass and vice versa.

```
17  
18 class CType_List : public std::list<const CType*>  
19 {  
20     ...  
21 }
```

22 CClass_Type holds a reference to CClass

```
23 class CClass_Type : CType  
24 {  
25     ...  
26     CType_List *m_pCtypeList;  
27     CClass *pCClass;  
28     ...  
29 }
```

```
30 // CClass holds a reference to CClassInfo and a reference  
31 to CClass_Type  
32 class CClass
```

```

1      {
2      ...
      CClass_Type *pCClassType;
      CClass_Info *pCClass_Info;
      ...
      }
3
      CClass_Info
4      {
      ...
      unsigned int uAssemblyToken;
      ...
      }
5
6

```

7 Metadata describes the types and classes in the portable code. Exemplary
8 metadata include: a name of the class; visibility information indicating the
9 visibility of the class; inheritance information indicating a class from which the
10 class derives; interface information indicating one or more interfaces implemented
11 by the class; method information indicating one or more methods implemented by
12 the class; properties information indicating identifying at least one property
13 exposed by the class; and events information indicating at least one event the class
14 provides.

15
16 The semantic analyzer 308 performs semantic analysis on the tokenized
17 parse tree 314. Semantic analysis involves traversing the tokenized parse tree 314
18 and validating types and operations with respect to the generic classes represented
19 in the parse tree. For example, the semantic analyzer 308 validates assignments
20 and casts with 'instanceof checks' to ensure that objects of generic classes are not
21 assigned to an invalid type. If invalid types or operations are identified by the
22 semantic analyzer 308, an error is generated during compile time. If no errors are
23 identified, the semantic analyzer 308 generates a validated tokenized parse tree
24 318.

25

1 The code generator 310 generates the compiled project 214 based on the
2 validated tokenized parse tree 318 and the generic classes 314. Code generator
3 310 converts the parsed and type checked tokens of the validated tokenized tree
4 318 into common intermediate language (CIL) code. The code generator 310
5 traverses the validated tokenized parse tree 318 gathering tokens. When the code
6 generator has enough tokens to create a line of CIL code, the corresponding CIL
7 code is appended to the portable code 216.

8
9 The code generator 214 creates the metadata 218 based on metadata in the
10 validated tokenized parse tree 318. The metadata 218 may be stored with the
11 project code 216 so that the compiled project 214 can be easily transported from
12 one platform to another platform. In addition, the metadata 218 can enable
13 another application program and/or developers to use the project code 216.

14
15 Although some exemplary methods and systems have been illustrated in the
16 accompanying Drawings and described in the foregoing Detailed Description, it
17 will be understood that the methods and systems are not limited to the exemplary
18 embodiments disclosed, but are capable of numerous rearrangements,
19 modifications and substitutions without departing from the spirit set forth and
20 defined by the following claims.